Toward Automatic Data Distribution for Migrating Computations

Lei Pan
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA
lei.pan@jpl.nasa.gov

Jingling Xue
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia
jxue@cse.unsw.edu.au

Ming Kin Lai, Michael B. Dillencourt, and Lubomir F. Bic
Department of Computer Science
University of California, Irvine
Irvine, CA 92697, USA
{mingl,dillenco,bic}@ics.uci.edu

ABSTRACT

Program parallelization requires mapping computation and data to processors. Navigational Programming (NavP), based on the principle of migrating computations, offers a different approach than the conventional solutions that use a SPMD model. The three major steps are: (1) Distribute the data, based on the access patterns within the sequential code; (2) Insert navigational commands (i.e., `hop` statements) in the sequential code for the computation to follow the data; and (3) Cut the sequential migrating thread and form the multiple resulting threads into a mobile pipeline. This paper focuses on data distribution. We introduce the *Navigational Trace Graph* (NTG), a mathematical structure that captures the alignment and distribution preferences of a sequential program. The data distribution is obtained by partitioning the NTG appropriately.

The advantages include: (1) Because NavP computations migrate freely across partitions, our methodology can focus exclusively on reducing communication overhead first and later determine the actual computation partition and parallelization. This is in stark contrast to SPMD, where the data partitioning imposes hard constraints on the threads because they are stationary; (2) We create a block cyclic distribution pattern that are uniquely suited for mobile pipelines to exploit full parallelism but not supported by the classical HPF; (3) Our approach aligns entries rather than dimensions of the arrays and thus captures more accurately the cost of data communication; and (4) Our solution allows the use of 1D arrays to store 2D or 3D data and supports sparse storage schemes.

Our methodology can be used either as part of an automated parallelizing compiler or as part of a human-aided parallelization effort. We provide visualization tools to support the latter scenario. We present experimental results from several scientific applications to demonstrate the effectiveness of our approach.

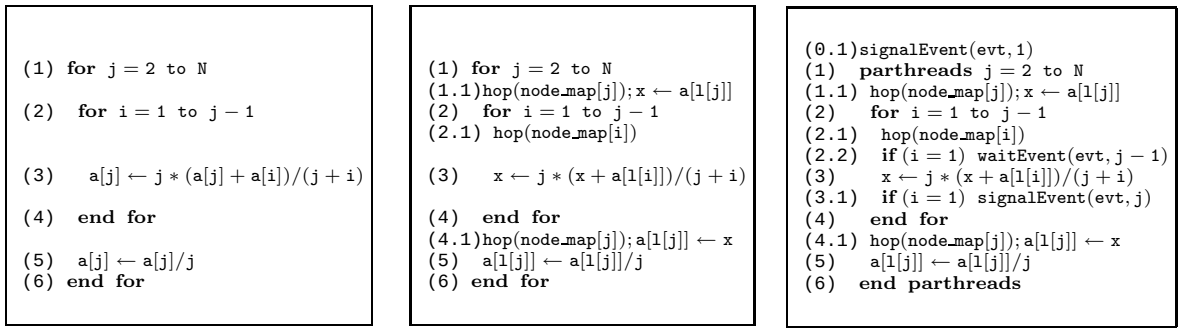
Keywords

data distribution, navigational trace graph, graph partitioning, navigational programming, migrating computations

1. INTRODUCTION

Distributed parallel programming is traditionally done in the Single Program Multiple Data (SPMD) style [19], in which a process or thread is stationary to a local partition of the data and thus is the owner of both the data and the computation associated with the data. Remote data that is required by a process is communicated by a `recv()` and a `send()` posted by the requesting and the owner processes, respectively. Navigational Programming (NavP) [29], which is the programming of self-migrating computations, is another means to distributed programming. The characteristics of NavP are: **(1)** Self-migration is made possible by explicitly inserting navigational statements (i.e., `hop(dest)`) into the code. The computation pauses at a `hop(dest)` statement, migrates to the destination processor `dest`, and resumes. Remote communication is achieved by threads carrying data from one location to another. Migrating threads are not permanent owners of any stationary data and the computation that each thread is responsible for is typically performed on more than one processor, incurring low cost in communication; **(2)** Self-migrating computations are user-level threads. They are non-preemptive and the synchronizations among them are through local events using the `signalEvent(evt)` and `waitEvent(evt)` statements. Two threads hopping between the same source and destination preserve a FIFO ordering; and **(3)** There are three kinds of variables: small data that follows a migrating computation is loaded to a *thread-carried variable*, while large data that is stationary to a processor is stored in a *node variable*. Multiple disjoint node variables can be used to construct a logical array spanning several processors, called a *Distributed Shared Variable (DSV)*. A DSV provides a partitioned global address space.

The NavP methodology provides four steps of code transformations. **Step 1. Data Distribution.** The input to this step is a sequential program to be parallelized. The objective is to find a data distribution that minimizes the cost of communication for the given sequential program, with a balanced (data) load as the constraint. What is being distributed is the large-sized data usually stored in a DSV. **Step 2. Sequential → DSC**. Using the data distribution obtained from Step 1, the sequential code is augmented with `hop()` statements to obtain a *distributed sequential computing (DSC)* program [24]. In a DSC program, there is a single thread that is responsible for the computation but auxiliary threads can be used for prefetching [24]. The optimization process of obtaining a good DSC is called DBLOCK *Analysis* [25], and it consists of identifying *Distributed Code Building Blocks (*DBLOCK*)* of appropriate granularities to resolve and then finding the right processor(s) to perform the computation for each DBLOCK resolved. A DBLOCK is a block of code that accesses data distributed across multiple processors [25]. In resolving a DBLOCK, we follow the *principle of pivot-computes* [29]. That is, the computation represented by a DBLOCK should take place on the processor, called a *pivot node*, that owns the largest portion of the distributed data. For a given data distribution, DBLOCK analysis essentially conducts computation distribution for NavP. **Step 3. DSC → DPC.** The DSC



(a) (b) (c)

Figure 1: A simple algorithm. (a) Sequential. (b) DSC. (c) DPC using mobile pipelining.

thread from Step 2 can be cut into several shorter DSC threads to build *mobile pipelines* [28] for *distributed parallel computing (DPC)*. The objective is to spread out computations as early as possible, respecting dependency requirements. A mobile pipeline can be phase-shifted to exploit full parallelism if the dependency relationship allows [29], achieving what the high-level language construct DOALL does. `signalEvent()` and `waitEvent()` are inserted to synchronize the DSCs constituting the mobile pipeline. **Step 4. Feedback loop.** This step estimates the tradeoffs between communication/parallelism and adjusts data distribution, DBLOCK analysis, and pipelining for a minimum overall wall clock time.

Achieving good data distribution is the topic of this paper. Good data distribution schemes are crucial to the performance of distributed parallel programs, because mistakes made in data distribution cannot be corrected by later programming efforts. We present a data decomposition approach and intend to use it as part of a data layout assistant tool for regular applications. The application programs that we are trying to help out thus are assumed to exhibit repeatable data accessing patterns – patterns that are seen in small-sized input data are going to show in very large problems. This assumption holds also for existing automatic data decomposition techniques for regular applications [2, 8, 11, 16, 20, 33, 22], since they need static or dynamic performance analysis to find out problem size parameters such as loop bounds and array sizes. As part of our tool, the programmer will be able to visualize the data layouts found and experiment with different input sizes. Irregular applications that are attacked using run-time solutions (e.g., the Adaptive Mesh Refinement technique) are outside the scope of this paper.

In comparison with existing automatic data decomposition techniques that are mostly analytical [2, 8, 11, 16, 20, 33, 22], our "numerical" approach has a number of advantages: **(1)** Because NavP computations migrate freely across partitions, our methodology can focus exclusively on reducing communication overhead first and later determine the actual computation partition into migrating threads and the parallelization using pipelining. This is in stark contrast to SPMD, where the data partitioning imposes hard constraints on the threads because they are stationary. Intuitively, since the mapping between a data element to the instances of the statements involving the element is one-to-many, deciding where the element goes is an easier problem than deciding how the computations are partitioned. Once a data distribution is chosen, computation partitioning can be done easily using migrating threads; **(2)** We create a block cyclic distribution pattern that is uniquely suited for mobile pipelines to exploit full parallelism but not supported by the classical HPF. This allows us to exploit full parallelism without needing to redistribution large amount of data as required by the DOALL approach; **(3)** We obtain data layouts by partitioning NTGs. So both alignment and distribution are solved very efficiently at the same time; **(4)** Our partitioning tool can find unstructured data layouts such as L-shaped blocks. These results are from regular algorithms that access structured data structures (e.g., dense square matrices), which means that unstructured data distribution is desirable even for seemingly simple applications. Our solution is able to do this because it aligns entries rather than dimensions of the arrays and thus captures more accurately the cost of data communication; and **(5)** Our approach is independent of array storage schemes used while those relying on component-affinity graphs (CAGs) or its variants [8, 11, 16, 20] are not. We can hence help the programs that use sparse storage schemes. We will justify these claims by examples and experimental results.

The rest of this paper is organized as follows. Section 2 introduces NavP using a simple example. Section 3 reviews the related work. In Section 4, we present our data decomposition technique in the context of turning sequential into DSC programs. Section 5 discusses how to find data distributions for DPC programs. Section 6 presents experimental results. Section 7 concludes the paper.

2. BACKGROUND

To make this paper self-contained, we use the following example to illustrate how NavP programming works. Consider the simple algorithm listed in Fig. 1(a), in which the j^{th} iteration of the outer loop, which computes `a[j]`, consumes the values of `a[i]` produced by all the previous `j` iterations. We assume a block data distribution pattern for simplicity, and use individual arrays on the PEs to host the data blocks. These arrays logically form a DSV. The auxiliary array `node_map[.]` provides the logical node hosting a given array entry, and `l[.]` contains the local array index of an entry with a given global index. A DSV thus provides a partitioned global address space. The computation of `a[j]` should take place on the PEs where the `a[i]`s reside, so that the cost of communication for the subcomputation of `a[j]` is minimized. We therefore put `a[j]` in a thread-carried variable, `x`, and insert `hop()` statements in the sequential code so that the computation follows the data it accesses (i.e., the `a[i]`s) through the network. The result is a DSC program: the computation uses distributed data but has a single locus of computation. Figure 1(b) shows the DSC code. Three `hop()` and load/unload compound statements are inserted (at lines (1.1), (2.1), and (4.1)) without changing the code structure. In the pseudocode, `x`, `i`, and `j` are thread-carried variables, and `a[.]` is a DSV. If we cut the single long DSC thread into multiple shorter threads, we get a DPC program, listed in Fig. 1(c). Each computation of `j` becomes a DSC thread that is spawned by the `parthreads` at line (1). The NavP `parthreads` construct generalizes the classical DOACROSS and DOALL parallelism constructs, but the spawned threads are DSC

threads. The code for each thread, lines (1.1) through (5), remains almost the same as the DSC code listed in Fig. 1(a). The only difference is the insertion of two new lines to synchronize the accesses to the entry a[1]. Each thread waits at line (2.2) until the previous thread is done accessing a[1], and at line (3.1) it notifies all other threads on the logical node that it has finished accessing a[1]. In this way, the threads organize themselves into a mobile pipeline when they access a[1]: the thread computing a[j] runs immediately after the thread computing a[j − 1]. Because of their FIFO scheduling, migrating threads do not pass each other in the mobile pipeline. Each computation migrates through the pipeline, progressively visiting the successive stages (the entries a[i] that it successively incorporates into its computation). Notice that in NavP synchronizations are only local among the collocated threads. Figure 2 schematically depicts how a mobile pipeline works.

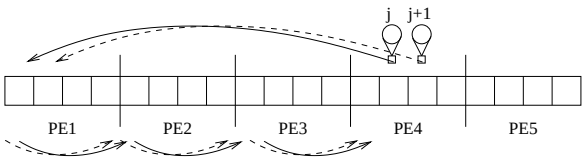


Figure 2: Mobile pipeline of DSC threads.

NavP has a number of advantages. **(1)** As validated by our preliminary results [29, 30, 28], NavP implementations are always competitive with the best MPI implementations in terms of performance, and in some cases are considerably better. As a special use of NavP, DSC threads can speed up the execution of even a single sequential process [24]. **(2)** NavP is structured distributed programming, as it directly captures the algorithm. MP, by comparison, requires significant restructuring of the program, obscuring its original purpose. It has been pointed out that `send` and `recv` are harmful today for much the same reason that unrestricted `goto` statements have been considered harmful since the "software crisis" of the 1960's [10]. Because of its structured programming, NavP allows us to parallelize certain programs that are generally considered unparallelizable using other approaches [28]. **(3)** NavP provides incremental parallelization, in the sense that a sequential program can be converted into a fully parallel program through a sequence of small transformations, where each intermediate step is a fully functioning program [30]. This is in sharp contrast to MP, where a parallel program usually requires a complete rewrite and major restructuring. **(4)** Today a hybrid programming model of MP+OpenMP is sometimes used [31], but this requires extensive programming efforts. NavP is a unifying approach that allows us to exploit both fine- (multithreading on shared memory) and coarse- (pipelined tasks on distributed memory) grained parallelism. These advantages make NavP a competitive alternative to MP as an intermediate representation for manual programming as well as automatic source-to-source code transformations by a compiler.

3. RELATED WORK

Distributed parallel computing has been a grand challenge, and it is perhaps more important today than ever as we reach hard physical limits in driving processor clock speed and therefore turn our focus on concurrency in software development. Recently, a number of new programming languages or systems have been proposed [4, 5, 14]. Good data distribution schemes are of key importance to all of them.

Compiling a sequential program for a distributed memory machine requires a decomposition (i.e., mapping) of the program's data and computation across the processors. In the SPMD model, data decomposition is performed first (by the programmer or compiler) while computation decomposition is inferred from the data decomposition using the owner-computes rule. In NavP, as discussed earlier in Section 1, DBLOCK analysis essentially conducts computation decomposition. So we will review only some automatic data decomposition techniques below.

Given a code region, which may be the entire program, two different approaches are distinguished: *static decompositions* [11, 20] (under which the data distribution for an array is fixed in the entire region) and *dynamic decompositions* [2, 8, 16, 33, 22] (under which different data distributions for an array may be used in different segments of the region). In the latter case, the region under consideration is divided into code segments, called *phases*, such that data remapping is only allowed between phases [8, 16]. In the former case, the entire region forms one single phase.

Given a phase, some techniques [16, 20, 33] decompose the mapping problem within the phase into two sequential steps: alignment and distribution. The alignment step identifies the dimensions of all arrays that should be mapped to the same dimension of a processor network. The distribution step decides which aligned dimensions should be distributed in a `BLOCK`, `CYCLIC` or `BLOCK-CYCLIC` manner. A central representation for the alignment problem is the weighted undirected *component affinity graph* (CAG) [20], where the nodes represent the dimensions of all arrays in the phase, edges the alignment preferences between dimensions of distinct arrays, and the edge weights the relative importance of alignment preferences. Alternatively, both steps can be solved at the same time. Based on the concept of CAG, both communication constraints (for reducing communication) and parallelization constraints (for preserving `DOALL` parallelism) are solved analytically [11] to find a data decomposition for a phase. In [2], affine transformations are used to find both data decomposition and computation decomposition simultaneously. They aim at computing communication-free mappings with the largest degree of `DOALL` parallelism [12].

There are several approachs to finding a data layout for a code region consisting of pre-defined phases. In [16], the problem is solved by 0-1 integer programming based on the so-called data layout graph (DLG), whose nodes are candidate layouts for every phase and edges represent the remappings between candidate layouts. The solution found is thus limited to the set of candidate layouts initially selected for each phase. Only 1D `BLOCK` distributions are considered. In [8], a variation of CAG, called *communication-parallelism graph* (CPG), is introduced so that alignment, distribution, remapping and `DOALL` parallelism are all considered in the same framework by 0-1 integer programming. In addition to 2D `BLOCK` distributions, 2D `CYCLIC` distributions are also considered for triangular loop nests. In [2], a so-called *communication graph* with its nodes representing the loop nests and edges the remappings between the loop nests is used to model data distributions that change dynamically. Both data and computation decompositions are found in a linear algebra framework in a greedy manner.

Like all these data decomposition techniques, our proposed technique is intra-procedural. Some inter-procedural techniques exist [3, 23]. Our technique is defined for individual phases, where a phase (to us) is a well-defined algorithm usually in the form of a function. This gives us the ability to handle large applications with multiple function calls. Extending our technique to multi-phase algorithms requires deciding whether or not to redistribute the data between each pair of consecutive phases. One approach is to apply our technique to each phase and to each sequence of consecutive phases, treating the sequence as a single phase. (This requires

applying our technique $O(n^2)$ times if there are n phases.) Once we have done this, the problem of deciding at which phase boundaries we should redistribute the data can be solved by a straightforward dynamic programming algorithm, quadratic in the number of phases. (The problem is essentially the same as finding a shortest path in a directed acyclic graph with positive costs on both edges and vertices.)

In comparison with the existing analytical techniques, our "numerical" approach has several advantages that we summarized at the end of Section 1. The problem of finding optimal data decomposition is known to be NP-complete. Previously, different techniques use different heuristics to estimate the benefits of parallelism and the cost of communication in their formulations. As discussed above, they find approximately optimal solutions analytically or by integer programming. Our approach is numerical in the sense that we find optimal solutions by using a graph partitioning tool (e.g., Metis [15]). Our approach is also approximate since such a partitioning tool is.

4. FINDING DATA LAYOUTS FOR DSC

When turning a sequential program into a DSC program, we must first find a data distribution for the DSC program. In this section, we present an intra-procedural technique for achieving this task. Presently, our technique works on individual phases, which are well-defined basic algorithms that are usually in the form of functions in scientific applications. In what follows, by a program we mean a phase (e.g., a code region) for which data layouts are to be found. How to find data layouts for multiple phases is our future work.

There are three key steps: **(1)** Build a so-called *navigational trace graph* (NTG) or trace graph for short by program instrumentation; **(2)** Find a data layout by partitioning the NTG using a graph partitioning tool; and **(3)** Express the data layout found using the data distribution mechanisms that NavP supports.

Let there be K processors. To find a data distribution for a DSC program, we will find a K-way partition of the corresponding NTG. The objective is to find such a data distribution by minimizing the cost of communication, with a balanced (data) load as the constraint. In Section 5, we will discuss how to find a cyclic data distribution for a DPC program with a balanced computation load. By using cyclic distributions, we can also make the tradeoffs between communication cost and exploitable parallelism.

4.1 Building an NTG

DEFINITION 1. An NTG for a program is a weighted undirected graph (without self-loops), where the vertices are the entries of DSVs (one for every entry of every DSV) and the edges (with positive weights) represent the affinity relations among the vertices as the locus of computation finds it way through them.

The NTG for a program is generated by running the program against a small problem. The larger the weight of an edge is, the stronger the two incident vertices want to stay together on the same PE. Unlike component-affinity graphs [8, 11, 16, 20, 21], our NTGs are constructed for NavP programs, which are equipped with mobility and can follow the data. Thus, the construction of an NTG follows the movement of the locus of computation under the NavP view [27]. In addition, the data entries of the arrays that will be distributed, regardless of which array they belong to, become the vertices of the same graph. In this way the problems of alignment and distribution, which are solved separately in other approaches, are addressed in a unified manner.

1 **Algorithm** BUILD_NTG
2 **INPUT:** a program
3 **OUTPUT:** a (weighted undirected graph) NTG $G = (V, E)$
4 Let *ListOfStmt* be a list of all statements executed in that order for a given program with respect to a small problem size
5 ***// Step 1: Edge Creation (with G being a multi-graph)***
6 Let V be the set of all DSV entries accessed in the program
7 Let $E = \emptyset$
8 *// Add L edges*
9 **for** every entry v in every DSV array
10 Add to E an L edge between v and each of its neighboring entries
11 *// Add PC edges*
12 **for** every statement s in *ListOfStmt* whose LHS is a DSV entry
13 Repeatly replace every non-DSV data entry v in the RHS of s, where v is defined by the statement of the form v = "...", with the "..."
14 Let RHS_s be the set of all DSV entries in the RHS of s
15 Add to E a PC edge between the LHS and every entry in RHS_s
16 *// Add C edges*
17 **for** every two statements s and t in *ListOfStmt* such that no statement in between in *ListOfStmt* has access to DSV data entries
18 Let V_s (V_t) be the set of all DSV entries accessed in s (t)
19 Add to E a C edge between every entry in V_s and every entry in V_t
20 Remove all self-loops in G
21 ***// Step 2: Edge Weight Selection***
22 #define L_SCALING = a nonnegative value (typically within $[0, 1]$)
23 Let *num_Cedges* be the total of C edges
24 Set $c = 1$
25 Set $p =$ *num_Cedges* $+ 1$
26 Set $\ell =$ L_SCALING $* p$
27 Merge the multiple edges linking the same two vertices into one single edge by accumulating their edge weights

Figure 3: An algorithm for building an NTG for a program.

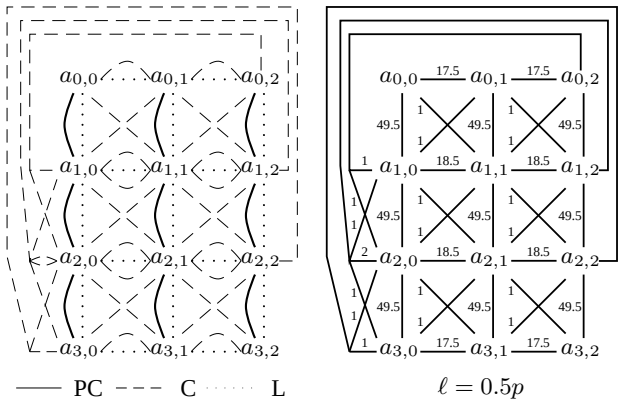
```
(1) for i = 1 to M − 1
(2)    for j = 0 to N − 1
(3)       a[i][j] ← a[i − 1][j] + 1
(4)    end for
(5) end for
```

Figure 4: A program for illustrating construction of NTGs.

An NTG is constructed in two steps: **(1)** edge creation, and **(2)** edge weight selection, which are described in the next sections.

4.1.1 Edge Creation

The construction of an NTG is based on three kinds of edges. First, locality (or L) edges are introduced between the neighboring entries of a DSV and they are assigned with the weight ℓ. These edges represent the locality of data access exhibited in many algorithms (thus they are not dependent upon the particular algorithm being parallelized), and they aim at obtaining regular data layouts for each array. Second, a producer-consumer (or PC) edge with the weight p is introduced between an LHS DSV array entry and a RHS DSV array entry. These edges indicate the occurrence of communication if the two linked entries do not reside on the same PE. Finally, every array entry of a DSV array in one statement is connected with every DSV entry in its successive (in time) statement with a continuity (or C) edge with the weight c. These edges represent the change of locus of computation (i.e., hops) if the two linked entries do not reside on the same PE and their purpose is to help improve the granularity of computation.



(a) NTG (after Step 1) (b) NTG (after Step 2)

Figure 5: NTGs for Fig. 4 (M=4, N=3).

Our algorithm given in Fig. 3 creates these edges in lines 5 – 20. In line 4, *ListOfStmt* is the list of all dynamically executed statements obtained by running the sequential program for a relatively small problem size. In lines 8 – 10, we introduce locality edges. In lines 11 – 15, we introduce PC edges, which represent data dependences among DSV entries. Note that a PC edge exists between two DSV entries if one depends on the other directly or indirectly via a chain of non-DSV data entries. Hence, line 13 is needed to detect these PC edges. Consider the following sequence of dynamically executed statements in a program:

```
...
t1 = b[3] + 1
t2  = a[2] + t1
a[5] = t2 + a[4]
...
```

where `a[]` and `b[]` are DSVs, and `t1` and `t2` are non-DSV entries. After line 13, `a[5] = t2 + a[4]` becomes

```
a[5] = a[2] + b[3] + 1 + a[4]
```

Thus, in lines 14 – 15, a PC edge is added between the DSV entry `a[5]` and each of the three DSV entries, `a[2], b[3]` and `a[4]`. After line 13, all the statements that define the non-DSV entries are ignored. It is possible to have multiple PC edges between the same two entries since the RHS entry may be fetched from its hosting processor multiple times. This can happen since the RHS entry is written multiple times and must be fetched each time before it is used. Even if the RHS entry is never updated, we may still choose to fetch it each time that it is used in order to obtain a scalable solution.

In lines 16 – 19, we add C edges to the NTG. Again, there may be multiple C edges between the same two entries representing multiple hops required if both do not reside in the same PE. In line 20, we remove all edges linking a vertex to itself.

At the end of this step, the NTG obtained is a multi-graph with possibly one L edge, multiple PC edges and multiple C edges between any two vertices. As an example, applying this part of our algorithm to the program in Fig. 4 yields the NTG shown in Fig. 5(a).

4.1.2 Edge Weight Selection

Given the roles that L, PC and C edges play, the relative magnitudes of their weights will be chosen such that if the weight of PC edges is $p = 1$, then the C edges will be assigned the weight of infinitesimal $c = \epsilon > 0$, and the L edges a nonnegative value $\ell \geqslant 0$.

The motivations for this weight assignment are as follows. As we shall see in Section 4.2, we obtain a data distribution for a program by partitioning its NTG such that the weights of the total cut edges are minimized. Since C edges have infinitesimal weights compared to PC edges, they cannot (and should not) collectively affect the producer-consumer affinity relationship of the data entries. Thus, C edge cuts are encouraged and so is parallelism because the C edges are not true dependences but artificial sequencing relations. As a result, the entries linked with PC edges tend to stay on the same PE. As for L edges, choosing different weights makes it possible to tradeoff between data locality and parallelism. If ℓ is close to p or larger, we will obtain a more regular partition, which usually results in better data locality. If ℓ is close to 0, the resulting data partition will reflect more accurately the actual cost of communication of the program. Such a partition tends to be less regular but may allow more parallelism to be exploited.

There can be more than one way of assigning edge weights. Our solution is given in lines 22 – 27 in Fig. 3, where `L_SCALING` is a program-dependent parameter, which can be tuned in the feedback loop of NavP based on performance profiling and evaluation.

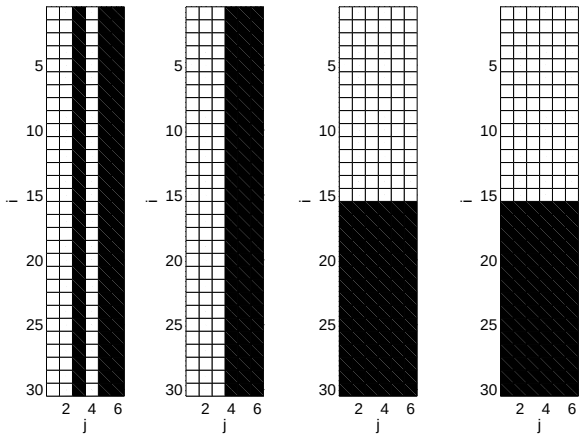
Figure 5(b) depicts the final NTG obtained for the program given in Fig. 4, under the assumption that $\ell = 0.5p$. The following section explains how such a graph is partitioned to obtain a data distribution for a machine configuration.

To understand the roles that L, PC and C edges play and our solution for their weight assignment (lines 22 – 27), let us consider the four partitions given in Fig. 6 for the example given in Fig. 4. In these (and all other) partition diagrams, all data entries sharing the same grey scale are assigned to the same partition. The NTGs for the example (with and without final edge weights) can be found in Fig. 5. Let us consider Fig. 6(a). When only PC edges are used, all array columns are not linked by any edges. A 2-way partition thus can contain any half of the columns. Such a partition exhibits full parallelism at the expense of some thread hops (i.e., fine grained computation). If we now include C edges and choose the weights of PC and C edges according to line 25, the C edges will play the role of tie-breakers and bring us a coarser grained data distribution shown in Fig. 6(b). Through edge weight selection, we prefer to cut all C edges rather than even a single PC edge when the NTG is partitioned. As a result, the data distribution obtained in Fig. 6(b) admits full parallelism with also a minimal number of hops. If we did not set edge weights using line 25, or in other words, if we set the C edges to be larger than infinitesimal compared to the PC edges, we might get the partition shown in Fig. 6(c) if the matrix is shaped long and thin. By introducing L edges, we will obtain more regular layouts, or precisely, block distributions if the weights of the L edges are chosen to be relatively large, as shown in Fig. 6(d). Compared to the first solution, the third and fourth solutions reduces the number of hops. Compared to the second solution, they lose some degree of parallelism; pipeline parallelism is exploitable but full parallelism is not since the computations on the two partitions cannot start simultaneously due to dependences within columns.

4.2 Partitioning the NTG

A DSC program consists of a single thread running on `K` PEs. An NTG will be fed to a graph partitioning tool to find a `K`-way partition with the overall objective of minimizing communication cost incurred by the partition under the constraint of a balanced (data) load.

Presently, the graph partitioning tool we use is Metis [15]. Metis



(a) PC edges (b) PC + C (c) PC + large C (d) PC + C + L ($\ell = 0.5p$)

Figure 6: Two-way data distributions obtained by a graph partitioning tool for the program given in Fig. 4 (M=50, N=4).

uses a parameter called UBfactor to specify the imbalance allowed between the partitions during recursive bisection [15]. If there are `n` vertices in the NTG, the number of vertices in each partition during each bisection step is between $(50 - b)n/100$ and $(50 + b)n/100$. In all the applications considered in this paper, UBfactor=1. In finding a `K`-way partition, Metis will minimize the sum of the weights of the cut edges spanning all `K` partitions. According to the Metis' web site, graphs with over 1M vertices can be partitioned in 256 parts in under 20 seconds on a Pentium Pro PC.

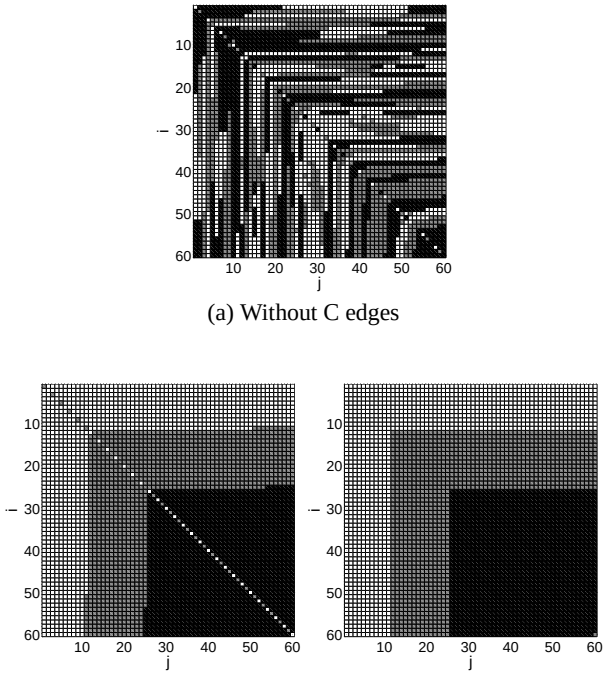
By finding a minimum cut to partition an NTG, we are able to minimize the total data movement among the PEs. We also maintain a data load balance in terms of data amount on the PEs because a balanced partition is used as an optimization constraint. However, balanced data load does not imply balanced computation load. This will not affect DSC since it runs in one thread. As a matter of fact, a balanced data load leads to a scalable DSC program. For DPC, we use block cyclic data distribution to achieve computation load balancing and better parallelism (more in Section 5).

Due to the presence of C edges in the NTG, which represent change of locus of computation, we minimize the number of thread hops. In other words, the C edges are helpful in keeping a coarse level granularity, which is important to performance. We introduce C edges to capture the artificial sequential dependency introduced in sequential algorithms. Our NTGs are generated such that cuts are more likely to be placed on the C edges to exploit parallelism, other things being equal. If cuts are on PC edges, they are more likely placed in the "direction" that is "parallel" to the PC edge chains because this results in less PC edge cuts. For this reason, we claim that our approach does not hinder parallelism.

4.3 Expressing the Partitions

We are building a visualization tool to present the recommended data layouts (i.e., partitions) to the programmers. The tool displays the partition, based on the mapping from array indices to graph vertices used in the code instrumentation. Our preliminary results are shown in Figs. 6, 7, 9, 11 and 12 as five examples.

These results are from regular algorithms that access structured data structures (e.g., dense square matrices), which means that unstructured data distribution is desirable even for seemingly simple



(a) Without C edges

(b) With C edges (c) With C+L edges ($\ell = 0.5p$)

Figure 7: Transpose of a 60x60 matrix (3-way partition).

applications. Therefore, NavP needs to support not only the classic distribution mechanisms such as `BLOCK` and `BLOCK-CYCLIC` as in HPF, `GEN_BLOCK` and `INDIRECT` mappings (limited to one-dimensional indirection arrays) as in HPF-2 but also others that can describe the unstructured data layouts found by a graph partitioning tool. How to describe unstructured data layouts in NavP will be part of our future work.

4.4 Applications

This section discusses show how our data distribution tool can be used to find data distributions in three important applications, matrix transpose, ADI (Alternating Direction Implicit) Integration, and Crout factorization. These applications, which exhibit different data access patterns, serve as good examples to validate our proposed methodology. For matrix transpose, our approach is able to find L-shaped communication-free data distributions that cannot be found by previously existing approaches [8, 11, 16, 20, 21]. We are able to find data distributions for ADI [17, 1, 16, 18] but do so by solving both alignment and distribution at the same time. As for Crout, the data distributions we find are independent of storage schemes used for arrays (unlike these previous approaches).

4.4.1 Matrix transpose

Matrix transpose swaps the anti-diagonal entries of a matrix. The pseudocode is omitted. The data distribution found as shown in Fig. 7 consists of L-shaped partitions; it is optimal in the sense that it is communication-free.

If we did not have C edges in the NTG, each anti-diagonal pair will still be distributed in the same partitions, but pairs will be distributed in a dispersed fashion, as shown in Fig. 7(a), unlike what is shown in Figs. 7(b) and (c), where contiguous partitions are seen.

With L edges (weight $\ell = 0.5p$), the resulting partition is regular (except that the bottom-right entry is included in the top-left partition), as shown in Fig. 7(c). In the absence of L edges ($\ell = 0$), the

partition becomes less regular, especially along the main diagonal of the matrix, as shown in Fig. 7(b).

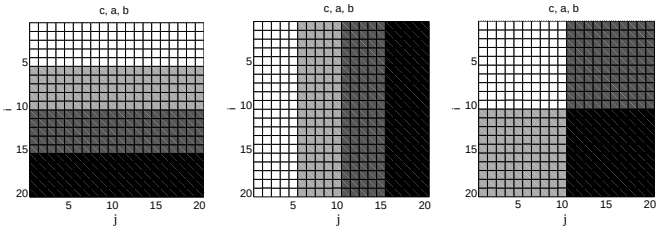
Our solution cannot be found by prior approaches since they are limited to BLOCK and BLOCK-CYCLIC [8, 11, 16, 20, 21]. This optimal solution enables the programmers to explore full parallelism with zero communication at a coarse granularity level.

4.4.2 ADI integration

```
// time iteration
(1) for iter = 1 to niter
      // Phase I : row sweep
(2)   for j = 2 to N
(3)     for i = 1 to N
(4)       c[i][j] = c[i][j] − c[i][j − 1] ∗ a[i][j]/b[i][j − 1]
(5)       b[i][j] = b[i][j] − a[i][j] ∗ a[i][j]/b[i][j − 1]
(6)     end for
(7)   end for

(8)   for i = 1 to N
(9)     c[i][N] = c[i][N]/b[i][N]
(10)  end for

(11)  for j = N − 1 to 1 by − 1
(12)    for i = 1 to N
(13)      c[i][j] = (c[i][j] − a[i][j + 1] ∗ c[i][j + 1])/b[i][j]
(14)    end for
(15)  end for

      // Phase II : column sweep
(16)  for j = 1 to N
(17)    for i = 2 to N
(18)      c[i][j] = c[i][j] − c[i − 1][j] ∗ a[i][j]/b[i − 1][j]
(19)      b[i][j] = b[i][j] − a[i][j] ∗ a[i][j]/b[i − 1][j]
(20)    end for
(21)  end for

(22)  for j = 1 to N
(23)    c[N][j] = c[N][j]/b[N][j]
(24)  end for

(25)  for j = 1 to N
(26)    for i = N − 1 to 1 by − 1
(27)      c[i][j] = (c[i][j] − a[i + 1][j] ∗ c[i + 1][j])/b[i][j]
(28)    end for
(29)  end for

(30)end for
```

Figure 8: Pseudocode of ADI

ADI integration is an example used by several papers on data distribution [17, 1, 16, 18]. The pseudocode for ADI is listed in Fig. 8 [17, 16]. There are three 2D arrays, namely c, a, and b, involved in the computation. This code is usually subdivided into two phases, namely a row sweep phase (lines (2)-(15)) and a column sweep phase (lines (16)-(29)). These two phases are surrounded by an outer loop of time iteration (line (1)). One possible solution, existed in previous work, is to find two different data mappings suited for their respective phases. We use our tool to find these two separate solutions and plot them in Figs. 9(a) and (b). Figure 9(c) depicts the data distributions for two phases combined together. The two sweeps are two DOALL loops (i.e., full parallelism with no communication) if they use their own data distribution, but in between the sweeps a dynamic data redistribution is needed. If both phases are combined, pipeline parallelism can still be exploited. The advantage of this data distribution for the entire program is that no dynamic data remapping is needed between the two phases. The cost of a dynamic data remapping can vary dramatically on different platforms.



(a) Phase I (b) Phase II (c) Phases I + II

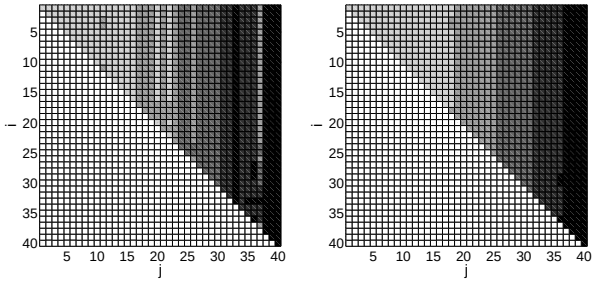
Figure 9: ADI integration on a 20x20 matrix (4-way).

4.4.3 Crout factorization

Crout factorization is a convenient variant of Gaussian Elimination [9, 13]. Figure 10 lists the sequential Crout algorithm. We assume that the matrix being factorized, K[], is a square and symmetric matrix. Since A is symmetric, only its upper half needs to be stored. In our implementation, K[] is a 1D array. When the matrix is sparse and banded, a 1D auxiliary array is used to store the index of the first non-zero entry (from the top) of each column.

```
(1)  for j = 1 to N
(2)    for i = 1 to j − 1
(3)      K[i][j] ← K[i][j] − Σ_{l=1}^{i−1} K[l][i] · K[l][j]
(4)    end for
(5)    for i = 1 to j − 1
(6)      T ← K[i][j]
(7)      K[i][j] ← T / K[i][i]
(8)      K[j][j] ← K[j][j] − T · K[i][j]
(9)    end for
(10) end for
```

Figure 10: Pseudocode of Crout factorization.



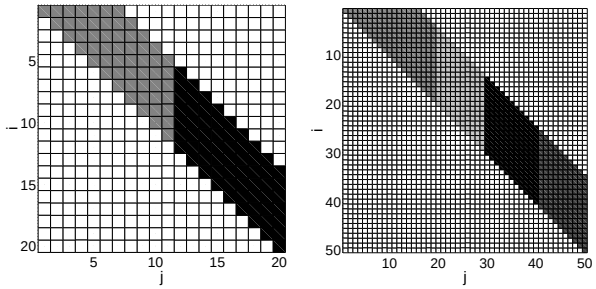
(a) Without L edges ($\ell = 0$) (b) With L edges ($\ell = p$)

Figure 11: Crout factorization on a 40x40 matrix (5-way).

As shown in Fig. 11, our tool suggests a column-wise partition. Note that the lower half of the matrix is not stored and should be ignored. For this algorithm, we obtain a regular data distribution if the weights of PC and L edges are chosen to be equal.

In our approach, an NTG is independent of the storage scheme used for the arrays in a program. This is an advantage over several other approaches [8, 11, 16, 20, 21] in which their component-affinity graphs are constructed from the dimensions of the matrices [18, 16, 7]. Furthermore, these approaches may have difficulty

in handling non-linear array subscript expressions introduced due to 2D-to-1D array storage mappings. As demonstrated here, our approach works when a 1D array is used to represent a 2D matrix. It also works for sparse and banded matrices (which are often stored in 1D arrays). Figure 12 shows two such examples.



(a) N=20, 2-way (b) N=50, 4-way

Figure 12: Crout factorization with sparse banded matrices (30% bandwidth).

5. FINDING DATA LAYOUTS FOR DPC

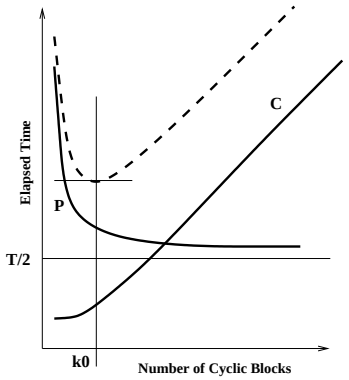


Figure 13: Performance as block cyclic data distribution is refined (assuming two PEs, T is the sequential execution time.).

In NavP, we parallelize a program by first transforming it to DSC and then turning DSC into DPC (Section 1). In Section 4, we presented our methodology for finding data layouts for DSC. The data partitions found do not hinder parallelism. Furthermore, they will also serve as the starting point to exploit more parallelism.

We propose to use a block cyclic data distribution evolved from the solution suggested by our tool and apply pipelining code transformation [26] to further improve performance. In our work, a block cyclic distribution means an `n`-round cyclic distribution of an (`nK`)-way partition to a `K`-processor machine, where the partitions can be rectangular or other shaped (e.g., L-shaped) blocks. So our block cyclic distribution is a more general form of `BLOCK-CYCLIC` distribution. It would be difficult, if not impossible, to find optimal block cyclic distributions automatically – there was not such a futile attempt before. In previous work on automatic data distribution techniques, `BLOCK` is exclusively used [16], `CYCLIC` is considered only for triangular loop nests [8]. When `BLOCK-CYCLIC` is used [11], the block sizes (or equivalently, `n`, here) are selected by an exhaustive search. In [2, 20], `BLOCK-CYCLIC` is considered only after data distributions have been found for virtual or parameterized processor spaces.

Figure 13 qualitatively depicts how the execution time changes as we refine the block cyclic data distribution to have smaller and smaller block sizes for the simple algorithm listed in Fig. 1. Our data distribution tool provides a partition with the minimum communication cost as our starting point (Number of Cyclic Blocks = 1 in Fig. 13). As we increase the number of cyclic data blocks, we obtain more and more parallelism (hence less and less time as depicted by the curve marked with P) at the cost of increased communication (depicted by the curve marked with C). Note that we follow the data distribution pattern suggested by our tool when we increase the number of data blocks (when the number of data blocks exceeds the number of PEs we call the data blocks "virtual blocks") – this will make sure that the communication cost remains the minimum for each and every new partition we come up with. At some point (when k=k0), the total execution time, depicted by the curve with dashed line, will reach the minimum and then start growing if we further increase the communication cost. Our proposed approach provides a systematic way of achieving the best performance for a particular application.

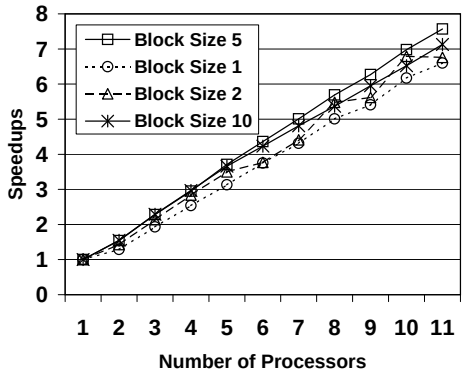


Figure 14: The performance of the simple problem.

As mentioned earlier, our data distribution guarantees data load balancing but not computation load balancing because we have migrating computations in NavP. Block cyclic data distribution is expected to help with computation load balancing because computations will migrate to all the PEs more evenly.

Performance data in Fig. 14 shows how adjusting block size of block cyclic data distribution could affect performance. When the block size of block cyclic data distribution is chosen to be 5, the performance is the best. A too coarse block size (of 10) or a too fine block size (of 1 or 2) gives us worse performance.

6. EXPERIMENTAL RESULTS

In this section, we present our experimental results. The data was obtained using a network of SUNW Ultra-60's with 450 MHz UltraSPARC-II CPU, 256MB of main memory, 1GB of virtual memory, 100Mbps of Ethernet connection with a collision-free switch, and using the NFS file-sharing system. The C compiler used was gcc 3.2.2, the MPI used was LAM MPI 7.0.6 [34], and the NavP compiler and runtime system used was MESSENGERS 1.2.05 [6].

6.1 Matrix Transpose

We have compared the costs of transposing a matrix in parallel under two circumstances: **(1)** Each PE gets a vertical slice of the

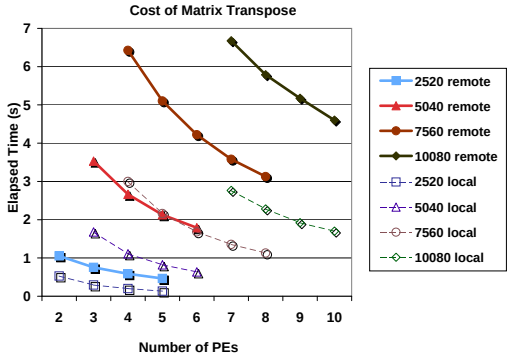
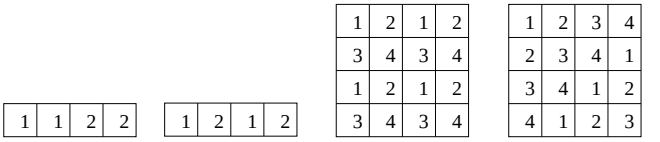


Figure 15: The cost of matrix transpose.



(a) 1D block (b) 1D blk cyclic (c) 2D blk cyclic (HPF pattern) (d) 2D blk cyclic (NavP pattern)

Figure 16: Block cyclic distribution patterns.

matrix, as depicted in Fig. 9(b). This data distribution requires remote data communication; and **(2)** Each PE gets an L-shaped slice of the matrix, as depicted in Fig. 7(c). Only local data movement is needed for this data distribution. Our experiment, as presented in Fig. 15, shows that matrix transposing involving remote communication is more than twice as expensive as done locally.

6.2 ADI integration

We first turn the ADI code into a block implementation. That is, we introduce "distribution blocks" — submatrix blocks that are basic units for data distribution — in the matrices and convert the loops over the matrix entries into the loops over the entries within the distribution blocks surrounded by the loops over the distribution blocks. Next, we go through the NavP steps to parallelize ADI. In particular, we first make the sweeps two DSCs and turn the outer loop another DSC responsible for injecting the sweeper DSCs. We then cut the sweeper DSCs into shorter ones and pipeline them. These steps are illustrated using the simple example presented in Section 2, we therefore skip the details here.

Figure 16 depicts two different block cyclic patterns — HPF and our own NavP — in 1D and 2D cases. Each box in this figure represents a submatrix block and the number in a box indicates the ID of the PE that this block is assigned to. It is assumed that in the 1D case we have two PEs and in the 2D case we have four PEs. As in Fig. 8, the three square matrices are each of order N. In Fig. 16(a), a matrix is cut into four vertical slices each of $N \times N/4$ and the blocks are assigned to the two PEs in a block fashion (that is, the first two blocks go to PE1 and the last two blocks go to PE2). Figure 16(b) depicts a 1D block cyclic pattern where the blocks are assigned to the PEs in order until the PEs are exhaustively used, at which time the block assignment cycles back. In HPF [32], a 2D block cyclic pattern is the cross product of two 1D block cyclic patterns, shown in Fig. 16(c). For 2D, each submatrix block is $N/4 \times N/4$. We develop our NavP block cyclic pattern, depicted in Fig. 16(d), in which the first row of blocks are assigned to all the PEs in order. (This is unlike the HPF pattern where the PEs are arranged as a 2×2 processor grid and the first row of blocks are assigned cyclically along the first row of processors.) The next rows are assigned to all the PEs in a similar way, except that they are shifted east-ward one position from their previous rows. This NavP block distribution is effectively a "skewed pattern." When the sweeper threads sweep through all the rows or columns, all PEs are busy simultaneously. That is, we achieve full parallelism, at the cost of $O(N)$ as one layer of the matrix entries is carried over from block to block. In contrast, in the example shown in Fig. 16(c), only two PEs are busy at any time as the sweeper DSCs sweep through. The situation for the HPF pattern is worse when the PEs are arranged as a 1D grid when, e.g., the number of PEs is a prime number. As for the cost of communication, the `DOALL` approach mentioned in Section 4.4.2 requires $O(N^2)$ in data redistribution.

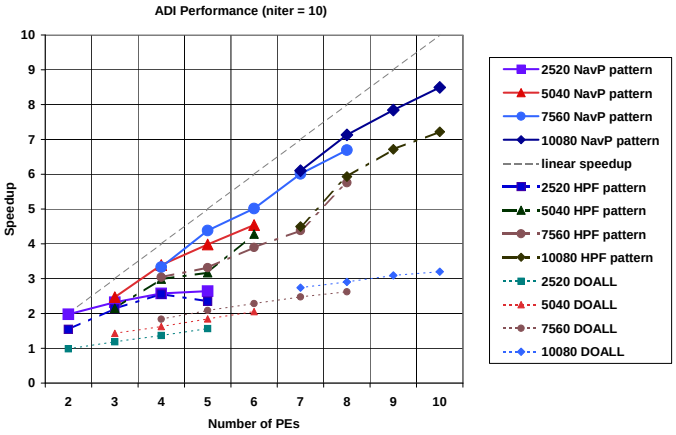


Figure 17: The performance of ADI.

As presented in Fig. 17 (the numbers in the legend are matrix orders), the NavP program using the NavP block cyclic data distribution pattern performs the best. Using the HPF block cyclic pattern, the NavP program incurs the same communication cost of $O(N)$ but has less degree of parallelism. Therefore, the performance is inferior, especially when the number of PEs is a prime number[1]. Finally, if we employ data redistribution in the `DOALL` approach, even though the two sweeps are fully parallel, the cost of data redistribution, $O(N^2)$, is so large that the overall performance is poor. We used the MPI library call `MPI_Alltoall()` to obtain the cost for matrix redistribution.

With this example of ADI, we are able to demonstrate the following: **(1)** We can solve both alignment and distribution, which are solved in separate steps in earlier work, in a unified manner; **(2)** The data distribution for NavP is obtained from minimizing the cost of communication with load balancing as a constraint. Parallelism is exploited later using mobile pipelines. The HPF style block cyclic data distribution helps to improve parallelism by making the PEs busy earlier, and NavP block cyclic data distribution, which is novel from this paper, enables the NavP program to achieve full parallelism; and **(3)** On loosely coupled systems such as clusters, data redistribution between the two phases, aimed at achieving full `DOALL` parallelism for both phases, is prohibitively expensive. As a result, choosing a data distribution that minimizes communication and further minimizing communication using DSCs that follow the principle of pivot-computes are of decisive importance to overall

[1]We use a true 2D processor grid for the HPF block cyclic pattern whenever possible.

performance. Using pipelining may result in loss of some degree of parallelism, but this impact to performance is secondary. Furthermore, with careful adjustment in data distribution using our NavP cyclic pattern, it is still possible to achieve full parallelism using mobile pipelines at a cost of asymptotically less communication than what is required in the DOALL approach.

6.3 Crout factorization

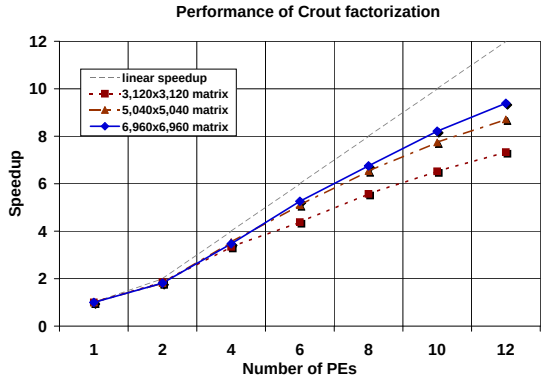


Figure 18: The performance of Crout factorization.

Crout factorization has the data access pattern similar to the simple example presented in Section 2, except that the problem is now 2D. We initially use the data distribution depicted in Fig. 11(b) and program our DSC thread to compute following the large-sized data like shown in Fig. 2. The difference is that the DSC now carries a column (entries on and above the diagonal line) of the 2D matrix rather than an entry of the 1D array. The DPC is obtained in the same way as described in Section 2 and block cyclic data distribution (using a block of columns as a distribution unit) is used to adjust the performance of the code as described in Section 5. We omit the details of implementation due to a space limitation and present the performance in Fig. 18.

7. CONCLUSIONS

This paper makes the following contributions:

- We present a new mathematical representation, called *navigational trace graph* (NTG), for representing the alignment and distribution preferences in a program at the level of DSV data entries in a unified manner. The NTG for a program is obtained by running the program against a small problem size. In a NTG, the nodes are the data entries of DSVs, the edges (classified as producer-consumer (PC) edges for dependences, continuity (C) edges for thread hops and locality (L) edges for layout regularity) represent the affinity relations between data entries, and the edge weights represent the relative importance of these affinity relations.

  One fundamental difference between NTGs and some previous representations such as communication-parallelism graphs (CPGs) [8] is that our NTGs do not impose explicit constraints for preserving all DOALL parallelism in the original program. However, our NTGs do not hinder parallelism because we carefully choose the weights of the edges such that the PC edges, which represent true dependency in the algorithm, are infinitely heavier than the C edges, which are from artificial sequencing of the program. More parallelism and load balancing are achieved by using block cyclic data distribution and mobile pipelining.

- We propose for the first time to use a graph partitioning tool as a general strategy to obtain a data distribution from a given NTG (for regular applications). Both alignment and distribution are solved very efficiently at the same time. Let there be K processors. To find a data distribution for a DSC program, we will find a K-way partition of the corresponding NTG. The objective is to find such a data distribution by minimizing the cost of communication, with a balanced (data) load as the constraint. For DPC, a cyclic data distribution in the form of an (nK)-way partition will be found, where n can be turned by performance analysis. By using block cyclic distributions, we can also make the tradeoffs between the communication cost and exploitable parallelism in a program.

- We create a NavP distribution pattern, effectively a skewed block data distribution, that allows us to exploit full parallelism without redistributing large amount of data as required by the DOALL approach. This pattern is novel and is uniquely suited for mobile pipelines to our best knowledge;

- Our partitioning tool can find unstructured data layouts such as the L-shaped blocks. It is able to do this because it aligns entries rather than dimensions of the arrays and thus captures more accurately the cost of data communication;

- Our approach is independent of array storage schemes used. We can hence help the programs that use sparse storage schemes.

- We present experimental results to show the effectiveness of our technique in three representative scientific applications.

Our future work includes extending this approach to large programs with multiple phases, developing an efficient algorithm to automatically recognize and capture the data distribution patterns in a given K-partition that human beings can recognize, and devising new language constructs that allow our programmers to express layouts that do not exist in other approaches.

8. REFERENCES

[1] Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam. Data and computation transformations for multiprocessors. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 166–178, Santa Barbara, California, 1995.

[2] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 112–125, New York, NY, USA, 1993. ACM Press.

[3] Jennifers M. Anderson. *Automatic Computation and Data Decomposition for Multiprocessors*. PhD thesis, Stanford University, Mar. 1997.

[4] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade high productivity language. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pages 52–60, 2004.

[5] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Vivek Sarkar, and Christoph Von Praun. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 519–538. ACM SIGPLAN, 2005.

[6] Department of Computer Science, University of California, Irvine, Irvine, Calif. *MESSENGERS User's Manual (Version 1.2.05 Beta)*, May 2005.

[7] Jordi Garcia, Eduard Ayguade, and Jesus Labarta. Dynamic data distribution with control flow analysis. In *Proceedings of supercomputing'96*, 1996.

[8] Jordi Garcia, Eduard Ayguadé, and Jesús Labarta. A framework for integrating data alignment, distribution, and redistribution in distributed memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 12(4):416–431, 2001.

[9] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, Md., third edition, 1996.

[10] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems*, 26(1):47–56, January 2004.

[11] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):179–193, 1992.

[12] C. H. Huang and P. Sadayappan. Communication-free hyperplane partitioning of nested loops. *J. of Parallel and Distributed Computing*, 19(5):90–102, 1993.

[13] Thomas J. R. Hughes. *The Finite Element Method : Linear Static and Dynamic Finite Element Analysis*. Prentice Hall, Englewood Cliffs, N.J., 1987.

[14] L. V. Kale and Sanjeev Krishnan. CHARM++: Parallel programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel programming using C++*, pages 175–213. MIT Press, 1996.

[15] George Karypis and Vipin Kumar. *h*METIS *A hypergraph partitioning package (version 1.5.3)*. Department of Computer Science & Engineering, University of Minnesota, Minneapolis, MN 55455, 1998.

[16] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4):869–916, July 1998.

[17] J. Knoop and E. Mehofer. Distribution assignment placement: Effective optimization of redistribution costs. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):628 – 647, June 2002.

[18] Peizong Lee and Zvi Meir Kedem. Automatic data and computation decomposition on distributed memory parallel computers. *ACM Transactions on Programming Languages and Systems*, 24(1):1–50, January 2002.

[19] Claudia Leopold. *Parallel and Distributed Computing: A Survey of Models, Paradigms, and Approaches*. John Wiley & Sons, New York, 2001.

[20] Jingke Li and Marina Chen. The data alignment phase in compiling programs for distributed-memory machines. *J. Parallel Distrib. Comput.*, 13(2):213–221, 1991.

[21] Angeles Navarro, Emilio Zapata, and David Padua. Compiler techniques for the distribution of data and computation. *IEEE Trans. Parallel Distrib. Syst.*, 14(6):545–562, 2003.

[22] Daniel J. Palermo and Prithviraj Banerjee. Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. In *LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 392–406, London, UK, 1996. Springer-Verlag.

[23] Daniel J. Palermo, IV Eugene W. Hodges, and Prithviraj Banerjee. Interprocedural array redistribution data-flow analysis. In *LCPC '96: Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, pages 435–449, London, UK, 1996. Springer-Verlag.

[24] Lei Pan, Lubomir F. Bic, and Michael B. Dillencourt. Distributed sequential computing using mobile code: Moving computation to data. In Lionel M. Ni and Mateo Valero, editors, *Proceedings of the 2001 International Conference on Parallel Processing (ICPP 2001)*, pages 77–84, Los Alamitos, Calif., September 2001. IEEE Computer Society.

[25] Lei Pan, Lubomir F. Bic, Michael B. Dillencourt, and Ming Kin Lai. Mobile agents – the right vehicle for distributed sequential computing. In Sartaj Sahni, Viktor K. Prasanna, and Uday Shukla, editors, *Proceedings, 9th International Conference on High Performance Computing - HiPC 2002*, volume 2552 of *Lecture Notes in Computer Science*, pages 575–584, Berlin, Germany, December 2002. Springer-Verlag.

[26] Lei Pan, Lubomir F. Bic, Michael B. Dillencourt, and Ming Kin Lai. From distributed sequential computing to distributed parallel computing. In C. Huang and J. Ramanujam, editors, *Proceedings of the 2003 ICPP Workshop on High Performance Scientific and Engineering Computing with Applications (HPSECA-03)*, pages 255–262, Los Alamitos, Calif., October 2003. IEEE Computer Society.

[27] Lei Pan, Lubomir F. Bic, Michael B. Dillencourt, and Ming Kin Lai. NavP versus SPMD: Two views of distributed computation. In Teofilo Gonzalez, editor, *Proceedings of the Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems*, volume 2, Algorithms, pages 666–673, Anaheim, Calif., November 2003. ACTA Press.

[28] Lei Pan, Ming Kin Lai, Michael B. Dillencourt, and Lubomir F. Bic. Mobile pipelines: Parallelizing left-looking algorithms using navigational programming. In *Proceedings, 12th International Conference on High Performance Computing - HiPC 2005*, Lecture Notes in Computer Science, Berlin, Germany, December 2005. Springer-Verlag.

[29] Lei Pan, Ming Kin Lai, Koji Noguchi, Javid J. Huseynov, Lubomir Bic, and Michael B. Dillencourt. Distributed parallel computing using navigational programming. *International Journal of Parallel Programming*, 32(1):1–37, February 2004.

[30] Lei Pan, Wenhui Zhang, Arthur Asuncion, Ming Kin Lai, Michael B. Dillencourt, and Lubomir Bic. Incremental parallelization using navigational programming: A case study. In *Proceedings of the 2005 International Conference on Parallel Processing (ICPP 2005)*, pages 611–620, Oslo, Norway, June 2005.

[31] Rolf Rabenseifner and Gerhard Wellein. Comparison of parallel programming models on clusters of SMP nodes. In H.G. Bock, E. Kostina, H.X. Phu, and R. Rannacher, editors, *In Modelling, Simulation and Optimization of Complex Processes (Proceedings of the International Conference on High Performance Scientific Computing, March 10-14, 2003, Hanoi, Vietnam)*, pages 409–426. Springer, 2004.

[32] Robert S. Schreiber. An introduction to HPF. *Lecture Notes in Computer Science*, 1132:27–44, 1996.

[33] Thomas J. Sheffler, Robert Schreiber, William Pugh, John R. Gilbert, and Siddhartha Chatterjee. Efficient distribution analysis via graph contraction. In *LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 377–391, London, UK, 1996. Springer-Verlag.

[34] Jeffrey M. Squyres and Andrew Lumsdaine. A component architecture for LAM/MPI. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Proceedings, 10th European PVM/MPI Users' Group Meeting*, volume 2840 of *Lecture Notes in Computer Science*, pages 379–387, Berlin, Germany, October 2003. Springer-Verlag.